Eideen Mozaffari

UID : 105988436

Project 3 Report

**Design**

*Board Class:*

I added one data member to record the number of holes on each side passed in by the user and another data member to initialize the beans in each hole. For the containers, I used two vectors, one for the North side and one for the South side, to store information about the state of beans on each side. I let index 0 in each container represent the pot, and let the remaining indices correspond to the holes on each side (i.e. 1 -- holes per side).

*Player Class:*

I added a data member to record the name of the player. For SmartPlayer, I added two private methods, evaluate and minimax, that are helper functions for the SmartPlayer’s choose move, which I will elaborate more on in the next section.

*Game Class:*

I added the following data members to create the game: a board, NorthPlayer, and SouthPlayer. I also added a data member to keep track of whose turn it is and another data member that knows if the game is over. I added a private method named sweep, which will sweep the remaining beans in play on the opponent’s side into their pot if the current player cannot make a move (i.e. the game is over).

**SmartPlayer Design**

For my SmartPlayer, I used two helper functions, evaluate and minimax, to help my SmartPlayer choose the right move. I’ll describe the functions working together for SmartPlayer’s chooseMove from the innermost-level to the outermost-level:

*evaluate:* I implemented 3 logic evaluation techniques when given in a board.

1. If the game is over, the current player cannot make a move, we evaluate the board by performing a “sweep” for the opponent, and then comparing the pots. We return positive infinity if South wins and negative infinity if North wins.
2. I also check for Board control (bean majority). What this specifically means is that if you sum all the beans in play and your own pot, if that sum is less than the beans in your opponent’s pot, you have lost. For example, if we sum all the beans in play on both the North and South sides and add it to North’s pot, if North has less beans in their pot than South has in their pot, North loses regardless of how the game continues, so we don’t need to search any further.
3. My last heuristic is simply returning the difference between North and South’s pot.

Since we want to maximize South and minimize North, we return very large numbers for South in a winning position, and very small numbers for North in a winning position. Otherwise, no winning position is found, so we just return the difference in pots.

*minimax*: I evaluate a board if SmartPlayer cannot make a move, the maximum depth has been reached, or the time limit for searching the Game Tree has been reached. I followed the pseudocode for minimax, adding in rules of Kahlah, such as getting another turn or a capture.

*choosemove:* I created a timer and local variables to call minimax, returning the best hole after minimax finishes that was passed in by reference to minimax.

Psuedocode

Board::Sow

If invalid initial hole passed in

Return false

else if (start on North’s Side)

While beans left to sow

current hole = initial hole passed in

while beans left to sow and current hole > 0

decrement current hole

if current hole is North’s pot

add bean to North pot, update beans left to sow,

update end hole

break

else (in North’s holes)

add bean to North hole , update beans left to sow

endhole is current hole

endSide is North

if beans left to sow

current hole is 1

while beans left to sow and current hole <= holes per side

add bean to South hole, update beans left to sow,

update end hole

increment current hole

endSide is South

else (start on South’s side)

While beans left to sow

current hole = initial hole passed in

while beans left to sow and current hole <= holes per side

increment current hole

if current hole is South’s pot

add bean to South pot, update beans left to sow,

update end hole

break

else (in South’s holes)

add bean to South hole , update beans left to sow

endhole is current hole

endSide is South

if beans left to sow

current hole is num holes per side

while beans left to sow and current hole > 0

add bean to North hole, update beans left to sow,

update end hole

decrement current hole

endSide is North

return true

Game::move

Set currentPlayer based on which side is passed in

If player cannot make a move

Sweep opponent beans

Game is over

Return false

Otherwise

Player chooses a move

Update the board

If player makes a capture

Move beans to their pot

While the player’s last been keeps in ending in their pot

If player cannot make a move

Sweep opponent beans

Game is over

Return false

Otherwise

Player chooses a move

Update the board

If player makes a capture

Move beans to their pot

Game is not over

Return true

SmartPlayer::evaluate

If the game is over

If South can’t make a turn

If North has more beans

Return -infinity

Else if South has more beans

Return +infinity

Else Tie

Return 0

If North can’t make a turn

If South has more beans

Return +infinity

Else if North has more beans

Return -infinity

Else Tie

Return 0

If the total beans in play + North’s pot < South’s pot

Return +infinity

If the total beans in play + South’s pot < North’s Pot

Return -infinity

Otherwise, just return South’s Pot – North’s Pot

SmartPlayer::minimax

If side is North

Set value to +infinity

Otherwise side is South

Set value to -infinity

If the player cannot make a move

bestHole = -1

evaluate board

return

If the timelimit is reached

bestHole. = -1

evaluate board

return

If max depth is reached

bestHole = -1

evaluate board

return

for each hole in holes

Set branch’s time limit

Save start time

Try to make the move

If capture

Set beans

Else if in land in pot, get another turn

If cannot make a move

Sweep opponent’s beans

Otherwise

Call minimax again to find best move

Create variables hole2, value 2

Call minimax for the opponent

If we found a better value for current player

Set better value

Set better hole

Update time left

Return

**Notable Challenges**

SmartPlayer was the toughest class to program. I spent the most time on it, and I’m still not sure if it returns the best possible move due to my limited time on testing. Implementing the timer for SmartPlayer was also tough, and I’m still not sure if I did it correctly, so I kept the depth proxy in there as well.

I’m not sure if calling minimax recursively when the player gets another turn is the correct thing to do, or if choosing a random hole to take the next turn is better. I chose to do the first of the two, because it made the most sense if we want to find the best possible move.

Also, I know I could’ve done something in Game::move to take advantage of the recursive nature of getting another turn after landing in your pot, but I couldn’t figure it out so, I left my duplicated code in there (I know the code could be cleaner).

Test Cases

void doBoardTests()

{

// The first set of Tests are provided in the spec by Professor Smallberg //////////////////////////////////////

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 &&

b.beans(SOUTH, POT) == 0 && b.beansInPlay(SOUTH) == 6);

b.setBeans(SOUTH, 1, 1);

b.moveToPot(SOUTH, 2, SOUTH);

assert(b.totalBeans() == 11 && b.beans(SOUTH, 1) == 1 &&

b.beans(SOUTH, 2) == 0 && b.beans(SOUTH, POT) == 2 &&

b.beansInPlay(SOUTH) == 3);

Side es;

int eh;

b.sow(SOUTH, 3, es, eh);

assert(es == NORTH && eh == 3 && b.beans(SOUTH, 3) == 0 &&

b.beans(NORTH, 3) == 3 && b.beans(SOUTH, POT) == 3 &&

b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

// ///////////////////////////////////////////////////////////////////////////////////////////////////////////////

Board b2(0, -1);

assert(b2.holes() == 1 && b2.totalBeans() == 0); // Test invalid for Board()

assert(b2.beans(NORTH, -1) == -1); // Test invalid for beans()

assert(b2.beans(NORTH, 2) == -1); // Test invalid for beans()

Board b3(6,4);

assert(b3.sow(NORTH, 0, es, eh) == false); // Test out of bounds

assert(b3.sow(SOUTH, 0, es, eh) == false);

assert(b3.sow(NORTH, 7, es, eh) == false);

assert(b3.sow(SOUTH, 7, es, eh) == false);

b3.setBeans(NORTH, 6, 0);

assert(b3.sow(NORTH, 6, es, eh) == false); // Test Empty Hole

b3.setBeans(SOUTH, 1, 0);

assert(b3.sow(SOUTH, 1, es, eh) == false);

assert(b3.sow(SOUTH, 2, es, eh) == true);

assert(es == SOUTH && eh == 6);

Board b4(2, 0);

b4.setBeans(NORTH, 1, 4);

b4.sow(NORTH, 1, es, eh); // Test start at North and end at North (loop around)

assert(es == NORTH && eh == 2);

b4.setBeans(SOUTH, 1, 5);

b4.sow(SOUTH, 1, es, eh); // Test start at South and end at South (loop around)

assert(es == SOUTH && eh == 1);

b4.setBeans(NORTH, 1, 9);

b4.sow(NORTH, 1, es, eh); // Two loops around

assert(es == NORTH && eh == 2);

b4.setBeans(SOUTH, 1, 10);

b4.sow(SOUTH, 1, es, eh); // Two loops around

assert(es == SOUTH && eh == 1);

assert(b4.moveToPot(SOUTH, 3, SOUTH) == false); // Test out of bounds

assert(b4.moveToPot(SOUTH, 3, NORTH) == false);

assert(b4.moveToPot(NORTH, 3, NORTH) == false);

assert(b4.moveToPot(NORTH, 3, SOUTH) == false);

assert(b4.moveToPot(SOUTH, 0, SOUTH) == false); // Test out of Bounds

assert(b4.moveToPot(SOUTH, 0, NORTH) == false);

assert(b4.moveToPot(NORTH, 0, NORTH) == false);

assert(b4.moveToPot(NORTH, 0, SOUTH) == false);

}

void doPlayerTests()

{

// The first set of Tests are provided in the spec by Professor Smallberg //////////////////////////////////////

HumanPlayer hp("Marge");

assert(hp.name() == "Marge" && hp.isInteractive());

BadPlayer bp("Homer");

assert(bp.name() == "Homer" && !bp.isInteractive());

SmartPlayer sp("Lisa");

assert(sp.name() == "Lisa" && !sp.isInteractive());

Board b(3, 2);

b.setBeans(SOUTH, 2, 0);

cout << "=========" << endl;

int n = hp.chooseMove(b, SOUTH);

cout << "=========" << endl;

assert(n == 1 || n == 3);

n = bp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

// ///////////////////////////////////////////////////////////////////////////////////////////////////////////////

HumanPlayer hp2("");

assert(hp2.name() == "" && hp2.isInteractive());

BadPlayer bp2("");

assert(bp2.name() == "" && !bp2.isInteractive());

SmartPlayer sp2("");

assert(sp2.name() == "" && !sp2.isInteractive());

Board b2(3, 0);

b2.setBeans(NORTH, 1, 3);

b2.setBeans(NORTH, 0, 3);

b2.setBeans(SOUTH, 1, 3);

b2.setBeans(SOUTH, 0, 3);

assert(bp2.chooseMove(b2, NORTH) == 1) ; // Only non-empty hole, except pot

assert(bp2.chooseMove(b2, SOUTH) == 1); // Only non-empty hole, except pot

Board b3(3,0);

assert(hp2.chooseMove(b3, NORTH) == -1); // Test Human player, no beans

assert(hp2.chooseMove(b3, SOUTH) == -1);

}

void doGameTests()

{

// The first set of Tests are provided in the spec by Professor Smallberg //////////////////////////////////////

SmartPlayer bp1("Bart");

SmartPlayer bp2("Homer");

Board b(3, 0);

b.setBeans(SOUTH, 1, 2);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 2);

Game g(b, &bp1, &bp2);

bool over;

bool hasWinner;

Side winner;

// Homer

// 0 1 2

// 0 0

// 2 0 0

// Bart

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 2 &&

g.beans(SOUTH, 1) == 2 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

// 0 1 0

// 0 3

// 0 1 0

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(NORTH);

// 1 0 0

// 0 3

// 0 1 0

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

// 1 0 0

// 0 3

// 0 0 1

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 1);

g.move(NORTH);

// 0 0 0

// 1 4

// 0 0 0

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 4 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == SOUTH);

// ///////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Replicate the endgame in spec

SmartPlayer bp3("Bart");

BadPlayer bp4("Homer");

Board b2(6, 0);

b2.setBeans(SOUTH, 0, 16);

b2.setBeans(NORTH, 0, 22);

b2.setBeans(SOUTH, 3, 1);

b2.setBeans(SOUTH, 4, 1);

b2.setBeans(SOUTH, 5, 1);

b2.setBeans(SOUTH, 6, 1);

b2.setBeans(NORTH, 1, 2);

b2.setBeans(NORTH, 2, 4);

Game g2(b2, &bp3, &bp4);

g2.play(); // SmartPlayer v BadPlayer

/\*

Homer

2 4 0 0 0 0

22 16

0 0 1 1 1 1

Bart

Bart chooses hole 6

Homer

2 4 0 0 0 0

22 17

0 0 1 1 1 0

Bart

Bart gets another turn.

Bart chooses hole 5

Homer

2 4 0 0 0 0

22 17

0 0 1 1 0 1

Bart

Homer chooses hole 1

Homer

0 4 0 0 0 0

23 17

1 0 1 1 0 1

Bart

Bart chooses hole 1

Homer

0 0 0 0 0 0

23 22

0 0 1 1 0 1

Bart

Homer has no beans left to sow.

Sweeping the remaining beans into Bart's pot.

Homer

0 0 0 0 0 0

23 25

0 0 0 0 0 0

Bart

Bart is the winner.

\*/

// Test if SmartPlayer can beat BadPlayer, regardless of starting side in a proper 6 hole 4 bean Kalah Board from scratch

SmartPlayer bp5("Bart"); // SmartPlayer on South side

BadPlayer bp6("Homer"); // BadPlayer on North side

Board b3(6, 4);

Game g3(b3, &bp5, &bp6);

g3.play(); // Smart Player wins

// I couldn't show the entire game, and it doesn't make sense to just copy a result, but this is the test I used for testing SmartPlayer in a real game.

BadPlayer bp7("Bart"); // BadPlayer on South side

SmartPlayer bp8("Homer"); // SmartPlayer on North side

Board b4(6, 4);

Game g4(b4, &bp7, &bp8);

g3.play(); // Smart Player wins

// I couldn't show the entire game, and it doesn't make sense to just copy a result, but this is the test I used for testing SmartPlayer in a real game.

}